View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Sheffield Hallam University Research Archive

XPL: A Language for Modular Homogeneous Language Embedding

Tony Clark

Department of Computer Science, Middlesex University, London NW4 4BT, UK

Abstract

Languages that are used for Software Language Engineering (SLE) offer a range of features that support the construction and deployment of new languages. SLE languages offer features for constructing and processing syntax and defining the semantics of language features. New languages may be embedded within an existing language (*internal*) or may be stand-alone (*external*). Modularity is a desirable SLE property for which there is no generally agreed approach. This article analyses the current tools for SLE and identifies the key features that are common. It then proposes a language called XPL that supports these features. XPL is higher-order and allows languages to be constructed and manipulated as first-class elements and therefore can be used to represent a range of approaches to modular language definition. This is validated by using XPL to define the notion of a *language module* that supports modular language construction and language transformation.

Keywords: domain specific languages, software language engineering, language modules

1. Modular Software Language Engineering

1.1. Background

There is increasing interest in *Software Language Engineering* (SLE) where new languages are defined as part of the system engineering process. In particular Domain Specific Language (DSL) Engineering [1, 2] is an approach whereby a new language is defined or an existing language is extended; in both cases DSLs involve constructing abstractions that directly support the elements of a single problem domain. This is to be contrasted with *General Purpose Languages* (GPLs) that have features that can be used to represent elements from multiple problem domains.

There are a large number of technologies that are currently used to engineer languages used in Software Engineering. These include: *grammarware* that are traditionally used to process syntax; *macros* that are used to extend a language with rewrite rules; language IDEs such as XText [3] and MPS [4] that can be used to generate language-specific tooling; TXL [5] and Stratego/XT [6] that are based on language rewriting.

Email address: t.n.clark@mdx.ac.uk (Tony Clark)

Some of these technologies have been available in languages that were designed many years ago. For example Common Lisp, designed in the 1980's contains a sophisticated macro system. Some of the tools are very new such as MPS and Stratego/XT. Emerging languages such as Fortress [7] include language extension mechanisms.

Although there are many different technologies that support SLE, they have many similar features. Many provide mechanisms for transforming text to syntax structures that are subsequently processed either in terms of evaluation or translation to other languages. Furthermore, many of these technologies allow new language features to be embedded in an existing language, effectively incrementally extending the host language. This form of host language embedding is called *homogeneous* because the new language features are *sugar* and do not require any new execution mechanisms compared to *heterogeneous* embedding that involves both new syntax *and* semantics.

This article aims to identify the key features of languages for SLE and to provide a language, called XPL, that supports these features. Unlike existing technologies for SLE, XPL is not concerned with pragmatic issues such as efficiency and tooling. Therefore, XPL provides a simple basis for the definition of SLE approaches and for designing new language features. To validate this claim, we use XPL to define a feature called *language modules* that can be used to build, analyse and transform new languages.

This article is constructed as follows: Section 1.2 provides an overview of existing SLE technologies and section 1.3 identifies common features. Section 1.4 reviews issues relating to *modular* homogeneous SLE and 1.5 describes the problems addressed and the contribution of this work. Section 2 introduces XPL and relates its features to homogeneous language embedding. Section 3 identifies different types of SLE modularity and shows how modularity can be encoded in XPL. Homogeneous language embedding in XPL is described in section 4 and is extended to module templates in section 5 and to module morphisms in section 6.

1.2. Technologies for Software Language Engineering

Languages can be defined using traditional compiler technologies such as Lex, Yacc, ANTLR [8], JavaCC [9], and more recent technologies such as XText [3]. These technologies provide mechanisms for defining grammars. A grammar defines the syntax of a language; grammar translations produce tools for language processing such as parsers and editors. Whilst most are mature, these technologies focus on syntax processing (although some provide technology for a limited form of static semantics). Pre-processor languages such as Awk and A* [10] can be used to implement new language features, however typically, a pre-processor has very limited knowledge of the underlying language structure and cannot access contextual information.

MetaML [11] was one of the first languages to identify *staging* in language processing where a multi-stage program involves the generation, compilation and execution of code, all within the same process. MetaML introduced four staging annotations that construct syntax, escape from quoted syntax, lift values to produce syntax, and run syntax values within a context. Template Haskell [12] used a similar approach and introduced monads to achieve hygiene.

Both MetaML and Haskell are statically typed so that compile-time meta-programming involves two type-checking phases: (1) code construction and manipulation is checked; (2) the resulting expression is spliced into the surrounding code before the entire program

is type-checked. The limitations of the approach are that the concrete syntax of both languages cannot be extended in any way and compile-time meta-programming cannot be nested.

The approach described in [13] reviews a number of languages that provide macros: the C pre-processor CPP; C++ templates; M4; Tex; Dylan; the Jakarta Tool Suite JTS; Scheme; the Meta Syntactic Macros System MS^2. Pre-processors, such as M4 and CPP, are described as being limited due to having no knowledge of the underlying syntax structure. The authors make a distinction between languages with one-pass and multi-pass macro systems and those with macro binding scopes. A distinction is made between macro calls within a macro definition that lazily expand (on each invocation) and those that eagerly expand (once at definition time). Macro systems differ in terms of the amount of error handling they provide, particularly in terms of trailing back from an error to the original syntax. The authors go on to define a macro language that allows new language constructs to be added to the host language in terms of syntax and metamorphism (syntax translation rule) definitions.

In [14] the authors propose a solution to what they term the *500 Language Problem* by which they mean the proliferation, and problems arising as a result, of the huge number of languages, public and proprietary, used in commercial software systems. The proposal is to base system engineering around a generic core and to use a grammar-based approach to provide interoperable technology for *language renovation*. The term *grammarware* is coined in [15] to describe an approach to Software Engineering in terms of the construction, tooling and maintenance of grammars, and by implication of a language driven approach to engineering software systems.

Language Oriented Programming [16] often involves extending an existing language with a macro system. Maya [17] is a system that supports language extensions to Java. New language constructs are added to the current Java language by defining *Mayans* that define grammar rules and how the rules synthesize language constructs. Each Mayan is defined in terms of pattern matching over existing Java language constructs. Maya provides access to the Java abstract syntax so that each Mayan can return an abstract syntax tree that is inserted into the surrounding tree. Maya supports hygiene by detecting variable binding and generating new variable names for locally bound variables in Mayans. OpenJava [18] is another example where meta-classes that inherit from OJClass implement a `translateDefinition()` method to expand occurrences of their instances (class definitions). OpenJava provides a limited form of syntax extension occurring at pre-defined positions in the Java grammar. Nemerle[1] is a language defined on the .NET platform that includes a macro definition system. Nemerle macros use quasi-quotes and drop-quotes, can be defined to be hygienic, can construct syntax and can extend the base language by defining new constructs in terms of existing constructs interleaved with newly defined keywords.

The Lisp family of languages provides macros for defining new language constructs. Lisp has an advantage when defining new syntax constructs because of the conflation of program and data into a single structure: the list. Common Lisp [19] macros are top-level definitions that use backquote (`` ` ``), comma (`,`) and comma-at (`,@`) to construct abstract syntax. The Scheme dialect of Lisp [20] provides similar features, however it goes further by providing syntax pattern-matching, hygiene and local syntax definitions.

[1] `http://nemerle.org`

Grammar composition can be used to define new languages in terms of old. The AHEAD approach [21] allows grammars to be refined by viewing them as being defined in terms of data members (tokens) and methods (syntax productions). Grammar composition in AHEAD produces the union of two grammars in terms of the sets of data members and methods where overlapping productions are controlled via a *run-super* mechanism. Grammar composition operators are described in [22] that allow an existing grammar to be extended with new productions, updates existing productions with new alternatives, and replaces an existing production so that all references are updated accordingly, and includes an early use of quasi-quotes for syntax construction. Attribute grammars can also be used to define extensible languages as described in [23]. A number of authors, including [24] and [25], describe formal properties of grammars that allow grammars to be compiled separately and the resulting combination of the parse tables do not lead to ambiguous languages.

New language constructs are often implemented in terms of a base language. Quasi-quotes are often used for this, but term rewriting is also used in systems such as Phobos [26] (which also uses inheritance between modular language definitions), Stratego/XT and ASD/SDF. An approach to grammar composition and evolution is described in [27].

1.3. Language Engineering Technologies: Key Features and Challenges

The previous section provides an overview of technologies for SLE. There are clear differences in approach and scope between the technologies. Our aim is to define a simple language (XPL) that represents the key features of SLE, study patterns of usage, propose extensions to them, and to devise new SLE features. In order to do this we identify the following key features required by SLE:

syntax definition All technologies for SLE allow new language features to be defined in terms of their concrete syntax. Typically this is achieved through the use of grammarware, although some languages use pattern matching over a uniform syntax, as in Lisp. Languages differ in terms of how the scope of the new feature is controlled. Many languages for SLE support only global language scope, although Scheme supports local definitions.

syntax synthesis Some languages for SLE allow syntax constructs to be explicitly synthesized. This can be done directly through the use of syntax constructors or indirectly through the use of quasi-quotes. When language features are embedded, the issue of variable capture must be addressed. Languages that encapsulate embedding are called hygienic [28].

syntax transformation Many languages for SLE allow syntax to be transformed. This can be done through syntax rewrite rules, or less abstractly through syntax accessors and constructors.

semantics New language features can be given a semantics by SLE technologies in two ways: translation to a target language that is different to the host technology (*heterogeneous*); translation or interpretation by the host technology (*homogeneous*). Heterogeneous language engineering is outside the scope of this article since it involves a target technology that cannot easily be incorporated into XPL.

embedding Homogeneous language embedding [29] involves a context switch when the host language uses an embedded language feature. This must address the issue of *scope* in terms of variables and flow of execution.

nesting Some SLE technologies allow language definition features to be nested. For example, Common Lisp macros can be written so that macro-expansion of a language feature produces a new macro.

expressivity Perhaps for practical reasons, SLE technologies are usually limited in terms of how languages can be combined and processed. For example, few technologies support parameterization over language definitions that would support language *templates*. As far as possible, XPL will impose few limitations on the definition and processing of language elements.

precision A major challenge for most SLE technologies is that they are defined with pragmatic considerations in mind. Therefore they are defined to integrate with IDEs and to run efficiently. Few technologies are precisely defined. An exception is PCF_{DP} [30] which is a calculus that addresses homogeneous language embedding in terms of quasi-quotes and embedding (referred to in PCF_{DP} as *unquote*). The language PCF_{DP} is concerned with allowing a λ-calculus to manipulate its own syntax in a type-safe way through `lift` and `drop` as defined in the rest of this article, although it does not address modularity issues or embedding different languages through syntax extension. This paper does not describe a formal semantics for XPL; it has been implemented as an interpreter in Java and validated through a series of examples.

1.4. Modular Homogeneous Language Embedding

As discussed in [31] language engineering can benefit from a modular approach in terms of *reuse* and *maintenance* providing that the modules are *self-contained*. It is also argued that the composition of languages must occur at both the syntax and the semantics level. The authors go further: *to realise self contained and reusable components it is vital to decouple reusable semantics of a component and semantics interconnection.*

There have been several attempts to develop approaches and technologies for modular language development. Many of the approaches are syntactic and therefore fall exclusively into the category of *grammarware* [32]. These include mechanisms such as Generalized-LR [33], Early parsing [34] and Packrat parsers [35] all of which aim to allow languages to be composed at the syntax level.

Different types of language design pattern are defined in [36] and further developed in [37] and [38]. These include:

piggybacking where a DSL uses features of a GPL in order to be a complete language. An example of this type of language is a state-machine DSL that uses C-syntax for an action language on transitions.

extension where a GPL is extended with features of a DSL in order to be tailored to a specific domain. An example of this type of language is adding SQL as a sub-language to Java.

specialization where a language is restricted by removing certain features in order to limit its use to the requirements of a particular domain. An example of this type of pattern is the restrictions placed on Ada to become SPARK Ada for use in real-time applications.

The MontiCore language [38] is designed to support the modular development of languages. This is achieved by having a language of grammars that can be composed in terms of multiple-extension. The semantics of the languages are defined in terms of the effects of the extension operators on class models of the synthesized abstract syntax structures.

Few technologies have been developed that address the composition of languages in terms of their semantics. Attribute grammars can encode the semantics of a language as part of the syntax definition, but [39] argues that this approach does not lead to a distinction between the component semantics and the composition semantics, and therefore is not modular.

Software libraries are argued to be language components in [40] where each library is given a DSL syntax front-end. This approach provides a useful mechanism for developing existing libraries as embedded DSLs but does not address the issue of semantics or composition. The authors do, however, raise an interesting issue of the scope of language embedding and define a number of different categories of scope.

All aspects of language components are discussed in [41] including syntax, type-checking, and operational features. The authors describe a system called Hive for expressing all these features which is similar to the Language Factories approach defined in [42], however neither of these systems give a precise definition of the composition mechanisms.

1.5. Problem and Contribution

The previous sections have reviewed tools for SLE and identified some common features. The representative tools and features are summarised in figure 1. It is argued in [39] that language engineering should be modular and that there are multiple ways in which language modules can be extended and combined [36, 38, 37]. However, few tools for SLE offer the precision and flexibility that would allow the design space of modular SLE to be explored.

This article takes a standard λ-calculus and extends it with features for SLE to produce a new language called XPL. In particular it provides support for syntax and semantics definition, embedding and abstract syntax processing. By making all new features first-class language concepts, XPL provides a flexible basis for exploring modular SLE. Figure 1 shows XPL as the final column where we claim that it provides most, but not all, of the key features. Those features not supported by XPL are argued as being important, but secondary to supporting SLE, since they can be added as extended features. Of course, the definition of many of these features is a research question and therefore it can be a matter of interpretation whether or not a given tool supports a given feature. Our claim is that XPL is more flexible than other tools; where we have claimed that XPL supports features such as syntax extension we mean that XPL is a suitable basis for constructing a range of approaches. The rest of this article defines XPL and provides concrete examples of modular language definition as evidence supporting this claim.

	XText	MPS	TXL	Stratego/XT	Lisp Family	Template Haskell	MetaML	MetaOCAML	JTS	Maya	OpenJava	Nemerle	Phobos	MontiCore	XPL
Syntax Synthesis The ability to create abstract syntax trees	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓
Syntax Transformation The ability to manipulate abstract syntax trees	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Heterogeneity Meta-language and defined language are different	✓	✓	✓	✓	×	×	×	×	✓	×	×	×	✓	✓	✓
Homogeneity Meta-language and defined language are the same	×	×	×	×	✓	✓	✓	✓	×	✓	✓	✓	×	×	✓
External Defined language is independent	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓
Internal Defined language is assimilated	×	×	×	×	✓	✓	✓	✓	×	×	✓	✓	×	×	✓
Templates Meta-language supports quasi-quotes	×	✓	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	×	✓
Hygiene Meta-language natively supports scoping	×	×	×	×	✓	✓	×	×	✓	✓	×	✓	×	×	×
Typechecking Target language typing is supported	×	✓	×	×	×	✓	✓	✓	✓	✓	✓	×	×	×	×
Tooling Target language tooling is supported	✓	✓	×	×	×	×	×	×	×	×	×	×	×	✓	×
Syntax Composition Multiple language syntaxes can be mixed	×	✓	✓	✓	×	×	×	×	✓	✓	×	×	✓	✓	✓
Nesting Multiple languages can be nested	×	✓	✓	✓	✓	✓	✓	✓	×	✓	×	✓	✓	✓	✓
Parameterisation Language definitions are the result of functions	×	×	×	×	✓	✓	✓	✓	×	×	×	×	×	×	✓
Piggybacking One language can be used within another	×	✓	×	✓	✓	✓	×	×	×	✓	×	×	✓	✓	✓
Extension Language definitions can be extended	×	✓	✓	✓	✓	×	×	×	×	×	×	✓	✓	✓	✓
Specialization Language definitions can be constrained	×	✓	✓	✓	×	×	×	×	×	×	×	×	✓	✓	✓
Semantics The semantics of the defined language is explicit	×	×	×	×	✓	✓	✓	✓	×	×	×	✓	✓	×	✓
Lift Values can be lifted to expressions	×	×	×	×	×	×	✓	✓	×	×	×	×	×	×	✓
Eval Language expressions can be transformed to values	×	×	×	×	✓	×	✓	✓	×	×	×	×	✓	×	✓

Figure 1: Representative SLE tools compared in terms of key features

2. An Extensible Programming Language (XPL)

XPL has been designed by taking a standard call-by-value λ-calculus and adding features that we hypothesize are characteristic of SLE technologies and are necessary to support modular homogeneous language embedding. An interpreter and read-eval-print-loop (REPL) for XPL has been implemented in Java. The implementation allows XPL definitions to be read from a file and subsequently referenced as part of user REPL interactions. All examples in this article have been implemented and are expressed as file-based definitions of values, `name = exp`, and functions, `name(args) = exp`, that are loaded before REPL interactions `exp` $\longrightarrow$ `value` where an expression is entered, evaluated and its value is printed.

The novelty in XPL is its execution mechanism and how that is used to encode modular languages. There is no formally defined type system for XPL. However, types can be useful in conveying the meaning of modular language semantics. Therefore, types are used as annotations (equivalent to comments) on XPL definitions in this article. Although the type language is used systematically, it lacks expressive power with regard to key aspects of XPL and is the subject of further work. Since a type system for XPL is not the main contribution, types are occasionally omitted when they are difficult to express using standard type language features. A basic understanding of functional programming and associated type systems is assumed.

This section introduces XPL by example. The basic features of XPL are described in section 2.1. Syntax construction and transformation are described in sections 2.2 and 2.3. XPL provides a sub-language for defining grammars; their use to process strings is described in section 2.4, and their use to define embedded languages is described in section 2.5 together with a series of example languages. The section concludes with a review of XPL as a basis for SLE in section 2.6.

2.1. Basic XPL

XPL is a λ-calculus with simple data structures: integer, string, bool, records and lists. This section provides a series of examples showing basic features. XPL keywords are in bold. The following shows the definition of a function that calculates the length of a list and the definition of a function that adds two-dimensional points that are represented using records:

```
length : ([t])->int
length(l) = if l = [] then 0 else 1 + length(tail(l))
length([0,1,2]) ⟶ 3
length(['zero','one','two']) ⟶ 3
length([true,false,true]) ⟶ 3

addvec : ({x:int;y:int},{x:int;y:int})->{x:int;y:int}
addvec(v1,v2) = { x = v1.x + v2.x; y = v1.y + v2.y }
addvec({x=1;y=2},{x=3;y=4}) ⟶ {x=4;y=6}
```

Each example above shows a definition with a preceding type annotation followed by a REPL evaluation. XPL provides three structured data types: lists and records (as shown above) and syntax trees (as defined in section 2.2). For convenience, records can be used as identifier bindings and placed in scope by the `import` expression, for example:

```
import {x=1} { x + 1} ⟶ 2
```

The record bindings override any bindings that are currently in scope, for example:

```
let r = {x=1}
    x = 2
in import r { x } ⟶  1
```

2.2. Syntax Construction

XPL provides constructors for building syntax. Each XPL language construct has a constructor that takes arguments corresponding to its sub-expressions. The constructors have names that correspond to the expression type that they construct and all start with a capital letter. To save space, the XPL language constructors are not all listed in this article and assumed to be obvious from the context of their use. An AST is decomposed using pattern matching in a `case`-expression. For example the constructors for if-expressions, binary-expressions, variables, and lists are used to create an expression that is then matched and decomposed using an if-pattern:

```
case If(BinExp(Var('l'),'=',List([])),Var('x'),Var('y')) {
  If(t,c,a) -> c
} ⟶ Var('x')
```

Working with constructors can lead to highly nested expressions. Therefore, a special syntax is provided called *quasi-quotes* (`[|` and `|]`) that are used to delimit *concrete* syntax expressions that are to be processed as though they are constructed using the AST constructors[2]. This is referred to as *lifting* the concrete expression to the syntax level:

```
case [| if l = [] then x else y |] { If(t,c,a) -> c } ⟶ [| x |]
```

As shown above, where possible, AST values are printed out using quasi-quotes by the XPL interpreter. An AST can be translated to a value using the `eval` operation. Since an AST might contain identifiers, the evaluation must be performed in the context of some identifier bindings that are supplied to `eval` as a record:

```
[| 10 |].eval({}) ⟶ 10
[| x |].eval({x=100}) ⟶ 100
[| if l = [] then x else y |].eval({l=[];x=1;y=2}) ⟶ 1
[| if l = [] then x else y |].eval({l=['dog'];x=1;y=2}) ⟶ 2
```

An AST is an expression that denotes an XPL value. If the AST `e` is evaluated to produce a value of type `t` then `e:[| t |]`. A value `v` of type `t` can be lifted to produce an expression `v.lift():[| t |]`. Lifting is a useful mechanism to move from the world of values to the world of expressions.

Quasi-quotes provides a convenient way of constructing ASTs. The examples shown above are *ground* in the sense that each AST expression has no variability. It is useful if we can include *holes* in the AST so that they can act as patterns or templates. Holes can be left in ASTs using *drop-quotes* (`${` and `}`) to surround an expression or literal that is referred to as being *dropped* into the surrounding AST. The following example shows how a pattern can be constructed using a function that can be used to generate field

[2]Note that constructors cannot be eradicated altogether because concrete syntax used in quasi-quote can only create *literal* atomic expressions such as identifiers and strings: `[| x |] = Var('x')`.

value accessors by mapping a field name to a function that references the value of the named field (where `t' ≤ {n:t}` denotes any record type `t'` that has at least one field named `n` whose type is `t`):

```
createAccessor : Fun(n:str) [| (t' ≤ {n:t}) -> t |]
createAccessor(name) = [| fun(record) record.${name} |]
createAccessor('x') ⟶ [| fun(record) record.x |]
createAccessor('y') ⟶ [| fun(record) record.y |]
```

XPL provides AST constructors for the entire language and provides quasi-quotes and drop-quotes as a convenience. The ability to freely construct ASTs is the basis for defining modular languages.

2.3. Syntax Transformation

Lift and drop can be used to transform expressions. The following are two simple examples of syntax transformers:

```
add1Exp : ([| int |]) -> [| [int] |]
add1Exp(x) = [| [ ${ x } + 1 ] |]
appExps : ([| [t] |],[| [t] |]) -> [| [t] |]
appExps(e1,e2) = [| ${ e1 } + ${ e2 } |]
```

The construction and transformation of non-trivial ASTs will often involve arbitrary collections of expressions. Lift and drop can be used to help. Consider the following definition of `foldr` that is used to process a list of elements (note that + is overloaded and used to append lists):

```
foldr : ((a)->b,(b,b)->c,b,[a])->[c]
foldr(f,g,b,l) =
  case l {
    [] -> b;
    h:t -> g(f(h),foldr(f,g,b,t))
  }
foldr(fun(x) [x+1],fun(l1,l2) l1 + l2,[],[100,200]) ⟶ [101,201]
```

Suppose that instead of a list of integers, we want to transform a list of expressions, each of which denotes an integer, into a single expression that denotes a list of integers where each integer has been incremented by `1`:

```
consolidate : ([ [| int |] ]) -> [| [int] |]
consolidate(exps) = foldr(add1Exp,appExps,[| [] |],exps)
consolidate([]) ⟶ [| [] |]
consolidate([ [| a |], [| b |] ]) ⟶ [| [a + 1] + ([b + 1] + []) |]
consolidate([ [| a |], [| b |] ] ).eval({a=100;b=200}) ⟶  [101,201]
```

The ability to define functions over ASTs is important for the modular construction of language expressions and for language transformation. Language *morphisms* are an important feature of SLE as described in section 6.

2.4. Grammars

XPL provides grammars that are essentially functions from strings to values. If a grammar `g` translates a string `s` to a value `v` of type `t` then `g` is of type `G(t)` and *synthesizes* `v` from `s`.

A grammar consists of a collection of rules, the first of which is designated the *start* rule. A *parse* describes the result of using the rules of a grammar to map a string to a value. The following is a very simple grammar:

```
g1 : G(int)
g1 = { start -> 'x' { 10 } }
g1.parse('x',[]) ⟶ 10
g1.parse('y',[]) ⟶ ERROR: expecting x
```

The grammar `g1` consists of a single rule called `start` that is also the start rule for the grammar. The body of the rule (after `->`) consists of two rule elements. The first `'x'` must match the prefix of the parsed string. The second element `{ 10 }` is an expression that produces a value synthesized by the parse. The value synthesized by the start rule of the grammar is the value returned by the parse.

A parse is performed by sending a grammar a `parse` message with two arguments. The first argument is the string and the second is a list of arguments to the start rule. In the example above, parsing `'x'` returns `10` and parsing `'y'` raises an error.

Rules may contain alternative elements separated by `|`. The following modification maps `'y'` to `20`:

```
g2 : G(int)
g2 = { start -> 'x' { 10 } | 'y' { 20 } }
g2.parse('x',[]) ⟶ 10
g2.parse('y',[]) ⟶ 20
```

Repetition is expressed by following a rule element with `*`. Since every rule element synthesizes a value, the element `e*` synthesizes a list of values each of which is produced by using `e` to consume 0 or more characters in the supplied string. For example, suppose we want to allow any number of `x` or `y`:

```
g3 : G([int])
g3 = { start -> (getx | gety)*;
       getx  -> 'x' { 10 };
       gety  -> 'y' { 20 } }
g3.parse('xyxyxyxy',[]) ⟶ [10,20,10,20,10,20,10,20]
```

The example above shows how multiple rules can be defined and how one rule can call another. We might want to allow white-space characters in the input string:

```
g4 : G([int])
g4 = { start   -> (getx | gety)*;
       getx    -> spaces 'x' { 10 };
       gety    -> spaces 'y' { 20 };
       spaces  -> (32 | 10 | 13 | 9)* }
g4.parse(' x    yxy  xy x      y',[]) ⟶ [10,20,10,20,10,20,10,20]
```

Finally, intermediate values may be required during a parse. Rules may have arguments that can be supplied in a rule element *call* as in `start^(n+m)` below. Identifiers may be bound to the values synthesized by rule elements as in `m=(getx | gety)` below. The following grammar adds up the values based on an initial value supplied as an argument:

```
g5 : G(int)
g5 = { start(n) -> m=(getx | gety) start^(n+m) | { n };
       getx     -> spaces 'x' { 10 };
       gety     -> spaces 'y' { 20 };
       spaces   -> (32 | 10 | 13 | 9)* }
g5.parse('',[0]) ⟶ 0
g5.parse('x',[0]) ⟶ 10
g5.parse('x x y x y',[0]) ⟶ 70
g5.parse('x x y x y',[9]) ⟶ 79
```

It can be useful to embed *semantic* tests during a parse. This is achieved using the `?` operator that constructs a rule element by applying a predicate to arguments. The arguments are evaluated with respect to the identifiers that are currently in scope including any bindings that have been constructed be preceding elements in the rule. The result is used to determine whether the current parse-alternative can continue. For example a range expression `[n,m]` is to synthesize a record `{low=n;high=m}` when `n < m`, is to just return `n` when the two values are the same and return `'error'` when `m < n`:

```
less  : (int,int) -> bool
eql   : (int,int) -> bool
range : G({low:int;high:int} + int + str)

less(n,m) = n < m
eql(n,m)  = n = m
range     = {
  start       -> '[' low=num ',' high=num ']' range^(low,high);
  range(n,m) -> ?less(n,m) { {low=n;high=m} } | ?eql(n,m) {n} | {'error'};
  num         ->  i=[48,57]+ { asInt(i) }
}

range.parse('[100,200]',[]) ⟶ {low=100;high=200}
range.parse('[100,100]',[]) ⟶ 100
range.parse('[100,20]',[]) ⟶ 'error'
```

Note that the rule `range(n,m)` above contains three alternatives. The first two are guarded by mutually exclusive conditions. The third alternative just synthesizes a string `'error'`. Alternatives in XPL grammar rules are tried in order left to right until one succeeds. XPL grammars are *compositional*. Given two grammars `g1` and `g2` the grammar `g1 + g2` contains all the rules from the component grammars and starts with the first rule of `g1`. Where rule names overlap in `g1` and `g2` the rules are merged as alternatives in the composite grammar with the `g1` rule taking precedence in a parse:

```
g6,g7,g8,g9 : G(int)
g6 = { start1 -> 'x' { 10 } }
g7 = { start2 -> 'y' { 20 } }
g8 = { start  -> 'x' { 10 } }
g9 = { start  -> 'y' { 20 } }
(g6+g7).parse('x',[]) ⟶ 10
(g6+g7).parse('y',[]) ⟶ ERROR
(g7+g6).parse('x',[]) ⟶ ERROR
(g7+g6).parse('y',[]) ⟶ 20
(g8+g9).parse('x',[]) ⟶ 10
(g8+g9).parse('y',[]) ⟶ 20
(g9+g8).parse('x',[]) ⟶ 10
(g9+g8).parse('y',[]) ⟶ 20
```

The quasi-quotes described in section 2.2 assume that the syntax of the language contained within the quotes is XPL by default. The default can be overridden by inserting a grammar `g` as follows `[ g | ... |]` where the form `[| ... |]` is equivalent to `[ XPL | ... |]`. The evaluation of a quasi-quote expression is as follows: evaluate `g`, use the resulting grammar to parse the text represented by `...` and delimited by `|]`, *lift* the resulting value and then evaluate the result. Note that *any* XPL value can be lifted to produce an expression. Since expressions are values, lifting an expression produces another expression. Lift is an identity operation on expressions of the form `${...}`. The following is a very simple example:

```
let g = { start -> 'add(' x=start ')' { [| ${x} + 1 |] }
                |  '0' { [| 0 |] }
                |  '<' x=XPL '>' { Drop(x) } }
    x = [| 2 + 3 |]
in [ g | add(add(<x>)) |]
⟶ [| ((2 + 3) + 1) + 1 |]
```

where `XPL` is the XPL grammar and `Drop` is the syntax constructor for expressions of the form `${...}`. The rest of this article will not require the use of quasi-quotes with non-default grammars.

XPL grammars map strings to values. Grammars themselves are first-class values that can be passed to functions, returned as results, stored in structures and composed. Both the synthesis of values and the tests performed during a parse can involve arbitrary XPL expressions. These features are all key to being able to represent, compose and transform language modules.

2.5. Languages

A grammar defines a language which is the set of strings that can be mapped to values by the grammar when it is viewed as a partial function. If the set of values produced by the grammar are ASTs then the grammar can be used to implement homogeneous language embeddings. In this case the grammar has the type `G([| t |])` for some type `t`. A language type is defined as: `L(t) = G([| t |])`.

In order for internal languages to work, XPL must provide a mechanism that places a grammar in-scope. This is done using `intern`:

```
xory : L(bool)
xory = { start -> 'x'* { [| true |] } | 'y'* { [| false |] } }
intern xory { xxxx } ⟶ true
intern xory { yyyyy } ⟶ false
```

The `intern` expression has two parts, the first is a grammar that defines a language and the second is a sentence in the language surrounded by `{` and `}`. Since grammars are values we can pass them to functions:

```
parse : (L(t))->t
parse(g) = intern g { xxx }
parse({ start -> 'x'* { [| true |] } }) ⟶ true
parse({ start -> 'x'* { [| false |] } }) ⟶ false
```

An `intern` expression supplies the text in its body as a string to the supplied grammar. The grammar must have a type of the form `L(t)` for some type `t`. The value that is synthesized by the grammar is therefore of type `[| t |]` and can be used as a replacement for the `intern` expression. If the synthesized expression contains identifiers then they will be resolved by the identifiers currently in scope[3]. Since the expression returns a value of type `t`, the type of the `intern` expression is also `t`. The rest of this section contains example language definitions.

[3]Note that we have placed hygiene out of scope for XPL taking the view that it is a restriction that can be added as a separate layer. If interned expressions contain unbound identifiers then this will be a run-time error.

2.5.1. A Language for Libraries

Grammars can be used as rule elements in which case the parse will call the start rule for the grammar. The value of the identifier `XPL` is the grammar for the XPL language and can be used to embed XPL, or parts thereof, within new languages. Here is a simple example that defines a library to be a header that describes what the library does, and a collection of definitions:

```
intern library {
  header
    This library performs arithmetic.
  end
  add(n,m) = n + m;
  sub(n,m) = n - m
} ⟶
{header='This library performs arithmetic.';
 defs={add=<closure>;sub=<closure>}}
```

The grammar uses the `XPL.fields` rule to parse the definitions:

```
library : ∀rec:RecordType L({header:str; defs:rec})
library = {
  lib -> h=head f=XPL.fields  {
    [| { header = ${ h.lift() }; defs=${ Record(f) } } |]
  };
  head -> 'header' s=nonWhitespaceChar* 'end' { asString(s) }
}
```

2.5.2. An Expression Language

The following grammar defines a simple multiplicative expression language. The grammar evaluates an expression as it is parsed and uses the XPL built-in operator `asInt` to map a sequence of integer character codes to an integer. This is an example of how XPL can be used to process *external languages* where the text is supplied as a string (possibly read from an external file):

```
arithExternal : G(int)
arithExternal = {
  start       -> a=atom tail^(a);
  atom        -> int | '(' a=start ')' { a };
  int         -> n=(['0','9'])+ { asInt(n) };
  tail(left) -> o=op right=start {
    case o {
      '*' -> left * right;
      '/' -> left / right
    }
  };
  tail(left) -> { left };
  op          -> ('/' | '*')
}
arithExternal.parse('10*20',[]) ⟶ 200
```

This article describes how XPL can be used to represent *internal languages* that are embedded within XPL. An internal language feature is defined by a grammar that synthesizes XPL abstract syntax:

```
arithInternal : L(int)
arithInternal = {
```

```
  start       -> a=atom tail^(a);
  atom        -> int | '(' a=start ')' { a };
  int         -> n=(['0','9'])+ { Int(asInt(n)) };
  tail(left)  -> o=op right=start {
    case o {
      '*' -> [| ${left} * ${right} |];
      '/' -> [| ${left} / ${right} |]
    }
  };
  tail(left)  -> { left };
  op          -> ('/' | '*')
}
intern arithInternal {10*20} ⟶ 200
```

XPL grammars, like those used by Template Haskell or MetaML, are based on a standard encoding using functional combinators [43] and cannot directly encode left recursion. However, XPL relies on the first-class and embedded features of grammars and therefore, providing these properties are preserved, the grammar language can be extended with sophisticated mechanisms such as that described in [44] for left recursion.

2.5.3. Processing Input Streams

Consider a business situation that needs to process serialised data[4]. This might occur when a company server must process a continuous data feed. Since the data is serialised, it is received as a sequence of character codes. The company wants to impose some structure on the data before it is processed. The following shows the required behaviour represented in XPL:

```
customer : ([int]) -> {customer:str;address:str;account:str}
customer =
  intern inflator {
    customer:5
    address:15
    account:3
  }
customer('fred 10 Main Road   501'.asList()) ⟶
{customer='fred ';
 address='10 Main Road   ';
 account='501'}
```

The language `inflator` is used to specify a string pattern. It is used to denote a function that maps a string to a record whose fields have the specified names and whose field values are the corresponding sub-strings.

The method `asList()` of a string returns the character codes in the string as a list. The `take` and `drop` list operators are used to define the `inflator` grammar:

```
take : ([t],int) -> [t]
take ([1,2,3],2) ⟶ [1,2]
drop : ([t],int) -> [t]
drop ([1,2,3],2) ⟶ [3]
```

An extractor, *e.g.*, `extractor('account',3)`, is a function that maps a sequence of character codes such as `'501xxx'.asList()` and a continuation `k` by supplying `k` with a record `{account='501'}` and the rest of the list `'xxx'.asList()`:

[4]Based on a scenario described by Martin Fowler: `http://www.infoq.com/presentations/domain-specific-languages`.

```
extractor('account',3)('501xxx'.asList(),k) ⟶ k({account='501'},'xxx'.
    asList())
```

The continuation `k` is used to process subsequent fields in the input and append `{account='501'}` to the subsequently constructed records. The definition of `extractor` is:

```
extractor : Fun(n:str) (int) -> [| ([int],({n:str},[int]) -> t) -> t |]
extractor(n,i) =
  let record = [| { ${n} = asString(take(l,${i})) } |]
  in [| fun(l,k) k(${record},drop(l,${i})) |]
```

Given a definition such as `customer` above, each field definition is translated into an extractor resulting in a list of extractors that are combined using `foldr` defined in the previous section together with the following operators:

```
combine(left,right) = [| fun(l) ${left}(l,fun(r,l) r + ${right}(l)) |]
id(x) = x
empty = [| fun(l) {} |]
```

Given these operators, the grammar simply maps the field definitions to extractors and combines them:

```
inflator : L(([int]) -> r)
inflator = {
  fields -> fs=field* { foldr(id,combine,empty,fs) };
  field -> n=name spaces ':' i=int { extractor(n,i) };
  int -> spaces n=numeric+ { Int(asInt(n)) };
  spaces -> (32 | 10 | 9 | 13)*;
  name   -> spaces l=alpha ls=alpha* { asString(l:ls) };
  alpha  -> ['a','z'];
  numeric -> ['0','9']
}
```

2.5.4. A Guarded Command Language

Suppose that we want to implement a guarded command language (GCL) and embed it within XPL. The embedding should allow XPL expressions to be referenced within the GCL when they are prefixed by $ and to return values to XPL. The value types are to be limited to booleans and integers. Here is an example of GCL used to implement the *greatest common divisor*:

```
gcd : (int,int)->int
gcd(n,m) :=
  intern gcl {
    a := $n
    b := $m
    do
      a < b -> b := b - a
      b < a -> a := a - b
    od
    abort a
  }
```

The semantics of such a language can be defined using a collection of constructors for the different types of language element. The GCL consists of language elements: programs; commands such as `update` and `do`; expressions; guarded commands of the form `e -> c`. The following types are used to annotate the element constructors:

```
type State   = [{name:str;value:int}]
type Cont    = (State)->int
type Fail    = () -> int
type Arm     = (State,Cont,Fail) -> int
type Command = (State,Cont) -> int
type Exp     = (State)->int
```

A program is constructed using the `begin` operator. A program state is represented as a list of records binding names to values. Since a command may modify the state, each command takes a state and a continuation, where the continuation represents what to do next and is supplied with the state once the command has completed. A program is supplied with a command and supplies it with an initial state and a final continuation (which by default returns `0` from the program):

```
begin : (Command)->State
begin(c) = c([],fun(state) 0)
```

Commands can be composed in sequence using the `seq` operator:

```
seq : (Command,Command)->Command
seq(c1,c2) = fun(state,cont) c1(state,fun(state) c2(state,cont))
```

The `update` operator constructs a command that changes the value of a variable. Since variable lookup will always use the first binding record, update is performed by adding a new record at the head of the state:

```
update : (str,Exp) -> Command
update(n,e) = fun(state,cont) cont({name=n;value=e(state)}:state)
```

Both conditional and loop commands are constructed from guarded commands (of type `Arm`). A guarded command is supplied with three arguments: a state, a continuation that is used if the guarded command is performed, and a fail that is used if the guard fails. In the case of a conditional command, the supplied continuation `cont` is performed whether the guard succeeds or not. In the case of the loop, if the guard succeeds the loop tries again (with an updated state) and if the guard fails then the loop terminates:

```
cond : (Arm) -> Command
cond(arm) = fun(state,cont) arm(state,cont,fun() cont(state))

do : (Arm) -> Command
do(arm) =
  fun(state,cont)
    arm(state,
        fun(state) do(arm)(state,cont),
        fun() cont(state))
```

To abort with a value, the program ignores the current continuation and returns the value of the supplied variable name (using `ref` defined below):

```
abort : (str) -> Command
abort(n) = fun(state,cont) ref(n)(state)
```

A guarded command is constructed from a boolean expression and a command. If the expression is true then the command is performed otherwise the fail continuation is called:

```
try : (Exp,Command) -> Arm
try(exp,command) = fun(state,success,fail)
  if exp(state)
  then command(state,success)
  else fail()
```

Two guarded commands are composed using `alt`. The second guarded command is supplied as the fail continuation to the first:

```
alt : (Arm,Arm) -> Arm
alt(a1,a2) =
  fun(state,success,fail)
    a1(state,
       success,
       fun() a2(state,success,fail))
```

Expressions are supplied with a state and return an integer. A variable reference finds the first binding record with the required name and returns the associated value:

```
ref : (str) -> Exp
ref(n) =
  fun(state)
    case state {
      r:s ->
        if r.name=n
        then r.value
        else ref(n)(s);
      [] -> '?' }
```

Two expressions are combined using `bin`:

```
bin : (Exp,str,Exp) -> Exp
bin(left,op,right) = fun(state)
  case op {
    '>' -> left(state) > right(state);
    '<' -> left(state) < right(state);
    '+' -> left(state) + right(state);
    '-' -> left(state) - right(state)
  }
```

A constant expression just returns the constant:

```
const : (int) -> Exp
const(k) = fun(state) k
```

The syntax of the language can be defined as an XPL grammar as follows:

```
gcl : L(int)
gcl := {
  program  -> cs=commands { [| begin(${cs}) |] };
  commands -> c1=command (c2=commands { [| seq(${c1},${c2}) |] } | { c1 });
  command  -> update | abort | cond | do;
  update   -> n=name spaces ':=' e=exp { [| update(${n.lift()},${e}) |] };
  abort    -> spaces 'abort' n=name { [| abort(${n.lift()}) |] };
  cond     -> spaces 'if' as=arms spaces 'fi' { [| cond(${as}) |] };
  do       -> spaces 'do' as=arms spaces 'od' { [| do(${as}) |] };
  arms     -> a1=arm (a2=arms { [| alt(${a1},${a2}) |] } | { a1 });
  arm      -> g=exp spaces '->' c=command { [| try(${g},${c}) |] };
  exp      -> a=atom (o=op e=exp {[| bin(${a},${o.lift()},${e}) |]} | {a});
  op       -> '>' | '<' | '+' | '-';
  atom     -> int | var | extern;
  var      -> n=name { [| ref(${n.lift()}) |] };
  int      -> spaces n=numeric+ { [| fun(state) ${Int(asInt(n))} |] };
  extern   -> spaces '$' e=XPL { [| fun(state) ${e} |] };
  spaces   -> (32 | 10 | 9 | 13)*;
  name     -> spaces l=alpha ls=alpha* { asString(l:ls) };
  alpha    -> ['a','z'];
```

```
  numeric  -> ['0','9']
}
```

GCL provides an example of a non-trivial executable language that is embedded in XPL. It demonstrates a number of features that are foundational to SLE; this section concludes with a review of this language in terms of these features.

In order for one language to be truly *embedded* in another, there must be some way to link the grammars so that one is aware of the other. GCL achieves this to some degree by referencing `XPL` in the `extern` rule. However this is not sufficient to achieve true embedding since `XPL` does not really know about `gcl`. There should be some way to achieve mutual dependence by allowing language definitions to be extensible and somehow *tying the knot* in the same way that mutually recursive procedures are defined.

Languages consist of syntax and semantics. GCL is a good example of a separation of syntax and semantics since it defines a collection of operators that represent the semantics and then the grammar uses these operators to construct the various language elements. However, to achieve composable, extensible language modules, the definition of syntax and semantics must be disciplined.

The next section reviews XPL as a basis for SLE and the rest of the article shows how XPL can be used in a disciplined way to achieve language modules consisting of both syntax and semantics that can be composed, transformed and extended.

2.6. XPL as an SLE Language

XPL is designed to be a simple, flexible, precisely defined language for expressing languages. Section 1.3 defines a list of features for any SLE technology that are reviewed in terms of XPL:

syntax definition XPL provides first-class grammars that are implemented using parser combinators as a record of higher-order functions that map text to XPL syntax.

syntax synthesis XPL provides syntax constructors, `lift` and `drop`. XPL is not hygienic since this property can be defined as a restriction on free syntax construction, but is not considered further in this article.

syntax transformation As noted above, XPL syntax is a first-class feature of the language. Pattern matching in `case` expressions can be used to transform syntax.

semantics Section 3 provides many examples where the semantics of embedded languages can be defined in XPL.

embedding XPL provides an `intern` expression that allows new language features to be embedded.

nesting In XPL, languages are first-class values and can be passed as arguments and returned as results. The normal rules of λ-scoping apply.

expressivity XPL inherits its expressivity from functional notations. Therefore, language features are not limited to the top-level of the language and can be parametric, transformed and composed.

3. Language Modularity

XPL has been designed as a basis for modular homogeneous language embedding. The previous section has introduced XPL. This section discusses a number of scenarios that language modularity must support (3.1), defines a basic encoding for a language module (3.2), defines a simple language module for expressions (3.3) and shows how language modules can be extended (3.4).

3.1. Modularity Scenarios

A *language module* is a unit of definition that provides the syntax and semantics for a language. Once it is defined, a language module can be used independently or transformed and combined with other language modules. The key modularity scenarios are as follows: (in each case we give references to language modules that are defined in the rest of the article):

independent An independent language module contains the syntax and semantics of a language, does not depend on the definition of any other language module and can be directly embedded in XPL using `intern`. The module `arithDirect` is independent.

substitution A language module contains separate definitions of syntax and semantics. A new language can be defined by substituting for one or other of these components. The modules `arithCalc` and `rational` are examples where the syntax of a base module is reused and the semantics are replaced.

extension A new language module can be defined as an extension of another by adding new language constructs. Section 5.2 provides a general pattern for defining extensible language modules using fixed points. This encoding is similar to that used to encode extension by inheritance in a functional language.

combination A new language module can be constructed by combining two independent modules. Languages in XPL are records containing higher-order functions that can be combined in various ways as described in section 5.1.

template A language can be defined in terms of one or more unknown features that are to be supplied as parameter values. A language module template (or *language factory*) is supplied with syntax or semantics features and returns a new language module. The template `arithOps` is defined twice: firstly requiring a predicate to define the names of infix operators; secondly requiring a grammar that defines the syntax of infix operators. Other templates include `arithExt`, `lambda`, and `bind`.

morphisms A language module morphism is a mapping from language modules to language modules. Generally, a morphism will require knowledge about the structure of its domain as in `addError`. It is possible to define language modules against a standard interface in which case morphisms are more general. Monads can be used to standardize the interface as in `arithM` and `rationalM` in which case a standard morphism such as `ND` can be applied.

3.2. Basic Language Modules

A language module is a pattern of language definition that separates syntax from semantics. In XPL this is done by constructing a record containing two fields. The first field is called `syntax` whose value is a function mapping a package of semantics to a grammar. The second field, called `semantics`, is a package of semantics definitions. A language is created by supplying `semantics` to `syntax:`

```
lang = {
  syntax(semantics) = {
    start...;
    ... grammar rules
  };
  semantics = {
    syntaxCnstr(args...) = ...;
    ... semantics for construct
  }
}
```

Language modules allow the syntax to be dependent on the semantics whilst allowing each to be independently transformed. In addition, language modules are a basic pattern whose variations, as we shall see, can be used to express different types of language definition.

3.3. An Independent External Language Module

The following XPL program code represents the arithmetic grammar from section 2.5.2 as a language module:

```
type ExpSemantics(t:Type) = { binExp:(t,str,t) -> t; int:([int]) -> t }
arithDirect : {
  syntax:(ExpSemantics(t))->G(t);
  semantics:ExpSemantics(int)
}
arithDirect = {
  syntax(semantics) = {
    arith   -> a=atom tail^(a);
    atom    -> i=int | '(' a=arith ')' { a };
    int     -> n=(['0','9'])+ { semantics.int(n) };
    tail(l) -> o=op r=arith { semantics.binExp(l,o,r) };
    tail(l) -> { l };
    op      -> ('/' | '*')
  };
  semantics = {
    binExp(left,op,right) =
      case op {
        '*' -> left * right;
        '/' -> left / right
      };
    int(cs) = asInt(cs)
  }
}
```

```
arithDirect.syntax(arithDirect.semantics).parse('10*20',[]) ⟶ 200
```

The grammar uses the semantics package to synthesize language values. Notice that the syntax is therefore independent of *how* the language is implemented.

3.4. Changing the Semantics: Substituting Calculations for Integers

Once a language has been expressed as a module, it is possible to change the meaning of denoted values by modifying or replacing the semantics package. Note that the interface provided by the semantics package must be preserved by any modifications since the grammar relies on these operations being available.

Suppose that we want to modify `arithDirect` so that the grammar synthesizes arithmetic *calculations* rather than integers. A calculation [45] is defined to be a tree that records all of the steps taken to evaluate the expression. A calculation has the form `{calc=l;value=n;children=cs}` where l is a label describing the step that was performed (either `'int'` for an integer-producing leaf-calculation or an operator: `'*'`; `'/'`). A new language module `arithCalc` is produced by reusing `arithDirect` as appropriate:

```
type Calc = { calc:str; children:[Calc]; value:int }
arithCalc : {
  syntax:(ExpSemantics(t))->G(t);
  semantics:ExpSemantics(Calc)
}
arithCalc = {
  syntax = arithDirect.syntax;
  semantics = {
    binExp(l,op,r) = {
      calc = op;
      children = [l,r];
      value = arithDirect.semantics.binExp(l.value,op,r.value)
    };
    int(cs) = {
      calc = 'int';
      children = [];
      value = arithDirect.semantics.int(cs)
    }
  }
}

arithCalc.syntax(arithCalc.semantics).parse('10*20',[]) ⟶
  {calc='*';
   children=[
     {calc='int';children=[];value=10},
     {calc='int';children=[];value=20}
   ];
   value=200}
```

The definitions of `arithDirect` and `arithCalc` show that by separating out the syntax and semantics, language modules allow syntax to be reused when defining new languages.

4. Homogeneous Language Embedding

XPL grammars are intended to be used to define both external and internal languages. An external language might use `parse` to transform a file containing the text of a language defined using XPL. An internal language uses `intern e1 { e2 }` which replaces the current grammar with the value of `e1` and then processes `e2`. The result of processing `e2` must be abstract syntax that is used by the underlying XPL execution framework in place of the `intern` expression.

XPL provides support for working with syntax as described in 2.2 in order to support internal languages. The following module is a version of the arithmetic expression language that uses syntax constructors.

```
arith : {
  syntax:(ExpSemantics([| t |]))->L(t);
  semantics:ExpSemantics([| int |])
}
arith = {
  syntax(semantics) = {
    arith    -> a=atom tail^(a);
    atom     -> i=int | '(' a=arith ')' { a };
    int      -> n=(['0','9'])+ { semantics.int(n) };
    tail(l) -> o=op r=arith { semantics.binExp(l,o,r) };
    tail(l) -> { l };
    op       -> ('/' | '*')
  };
  semantics = {
    binExp(left,op,right) = BinExp(left,op,right);
    int(cs)               = Int(cs)
  }
}
intern arith.syntax(arith.semantics) {10*20} ⟶ 200
```

The embeddable language module above can be transformed into one that supports rational numbers. A rational number is represented as a value `{num=n; den=d}` where `n` and `d` are numerator and denominator respectively. The syntax definition remains the same as given in `arith` and the semantics are changed to create appropriate rational number expressions:

```
type Rational = { num:int; den:int }
rational : {
  syntax:(ExpSemantics([| t |]))->L(t);
  semantics:ExpSemantics([| Rational |])
}
rational = {
  syntax = arith.syntax;
  semantics = {
    binExp(left,op,right) =
      case op {
        '*' -> [| { num=${left}.num * ${right}.num;
                    den=${left}.den * ${right}.den } |];
        '/' -> rational.semantics.binExp(left,'*',
                  [| { num=${right}.den; den=${right}.num } |])
      };
    int(cs) = [| { num=${Int(cs)}; den=1 } |]
  }
}
intern rational.syntax(rational.semantics) {10/3} ⟶ {num=10;den=3}
```

This demonstrates that language modules can support reuse in terms of a language whose syntax is of type `L(t)` and whose semantics is a package of functions that generate an AST of type `[| t |]`.

5. Language Module Templates

A language module is a record that conforms to a pattern and as such is a value that can be passed as an argument to a function and returned as a result. Together with lexical scoping rules, this makes language modules a powerful abstraction mechanism for engineering languages. Features of a language can be abstracted as arguments to a function that returns a language module; this is an example of a language factory [42] and a simple example is shown in section 5.1. Language modules can be used to represent extensible languages by abstracting over an extension point. An extensible language module of this kind is instantiated by finding its fixed point. Section 5.2 shows how fixed points are constructed in XPL.

5.1. Parametric Languages

In the arithmetic language modules defined in previous sections we have fixed the collection of operators: `*` and `/`. Suppose that we want to set up a language factory that generates language modules where the languages differ in terms of the operators that can be used. The following defines a function `arithOps` that receives a predicate `isOp` that tests whether an operator is legal in the resulting language module:

```
arithOps : ((str)->bool)-> {
  syntax : (ExpSemantics([| t |]))->L(t);
  semantics : ExpSemantics([| int |])
}
arithOps(isOp) = {
  syntax(semantics) = {
    arith      -> a=atom tail^(a);
    atom       -> i=int | '(' a=arith ')' { a };
    int        -> n=(['0','9'])+ { semantics.int(n) };
    tail(left) -> o=. ?isOp(asString([o])) right=arith {
      semantics.binExp(left,asString([o]),right) };
    tail(left) -> { left }
  };
  semantics = arith.semantics
}
module1, module2 : {
  syntax : (ExpSemantics([| t |]))->L(t);
  semantics : ExpSemantics([| int |])
}
module1 = arithOps(fun(o) o = '*' or o = '/')
intern module1.syntax(module1.semantics) {10*20} ⟶ 200
module2 = arithOps(fun(o) o = '+' or o = '-')
intern module2.syntax(module2.semantics) {10+20} ⟶ 30
```

A more expressive way of achieving a similar result is to pass entire grammars as arguments. Like `intern`, when a grammar is called, its starting non-terminal is taken to be the first rule in its definition:

```
arithOps : (L(str)) -> {
  syntax : (ExpSemantics([| t |]))->L(t);
  semantics : ExpSemantics([| int |])
}
arithOps(op) = {
  syntax(semantics) = {
    arith      -> a=atom tail^(a);
    atom       -> i=int | '(' a=arith ')' { a };
```

```
    int         -> n=(['0','9'])+ { semantics.int(n) };
    tail(l)     -> o=op r=arith { semantics.binExp(l,o,r) };
    tail(l)     -> { l }
  };
  semantics = arith.semantics
}

op1, op2 : G(int)
op1 = { op -> '*' | '/' }
op2 = { op -> '+' | '-' }

module3, module4 : {
  syntax : (ExpSemantics([| t |]))->L(t);
  semantics : ExpSemantics([| int |])
}
module3 = arithOps(op1)
module4 = arithOps(op2)

lang1, lang2 : L(int)
lang1 = module3.syntax(module3.semantics)
lang2 = module4.syntax(module4.semantics)
intern lang1 {10*20} ⟶ 200
intern lang2 {10+20} ⟶ 30
```

In the definition given above there are two grammars `op1` and `op2` defined as extension points for the language factory `arithOps`. The factory is instantiated twice to produce `module3` and `module4` that can then be transformed into languages `lang1` and `lang2`. These languages can be used independently or merged using +. A grammar that is produced by combining two base grammars in this way contains all the base rules in the same order that they are defined in the left and then the right grammars. Grammar rules with the same names in each operand grammar are merged as alternative definitions. This leads to the following:

```
intern lang1+lang2 { 10 * 20 } ⟶ 200
intern lang1+lang2 { 10 + 20 } ⟶ 30
intern lang1+lang2 { 10 * 20 + 30 } ⟶ Error!
```

Notice that in the last case we do not get the desired result. This is because the grammar that is processing operators is passed as an argument and, in the case of `lang1` we cannot have `+` and in the case of `lang2` we cannot have `*`, therefore they are mutually exclusive. The grammars have been merged at the wrong level, we want to give rise to options over the operators:

```
intern arithOps(op1 + op2).syntax(arithOps(op1 + op2).semantics) {
  10 + 20 * 30
}⟶ 610
```

This is the subject of the next section.

5.2. Extensible Languages

Often a language factory defines a basis for extension or variation. In the case of `arithOps` in the previous section, the variation point is passed as an argument grammar and does not need to know about any of the constructs in the base language. However, it is more useful if a factory can be parameterized with respect to language constructs that,

when they are supplied, can contain constructs from the base language *without knowing what they are in advance.*

To achieve such incremental and mutually recursive extensions we can use fixed points over language modules. The general structure is as follows:

```
lang = {
  syntax(root,extension,semantics) = {
    start -> ...;
    ... grammar rules defined in terms ...
    ... of root, extension and semantics ...
  };
  semantics = {
    syntaxCnstr(args...) = ...;
    ... syntax constructors
  }
}
```

The `root` element is a grammar that can parse the extended language, `extension` is a grammar that parses new language constructs, and `semantics` is a package of semantics operators as before. The result of the `syntax` function is a grammar that processes a basic language but which uses `root` and `extension` in specific ways: `root` used when the grammar wishes to process a root-construct, it is used to allow the language to be recursive; `extension` is used when the grammar wishes to grant an extension point. The extension may fail if the language has not been extended.

In the case of arithmetic expressions, the root-element is any integer or binary expression. Therefore, assuming for convenience that arithmetic expressions are right associative, the root element `exp` is used to parse the right operand of a binary expression. The extension point is used as an alternative type of expression:

```
spaces : G([int])
arithBase : {
  syntax : (L(t),L(t),ExpSemantics([| t |])) -> L(t);
  semantics : ExpSemantics([| int |])
}

spaces = { spaces -> (32 | 10 | 9 | 13)* }

arithBase = {
  syntax(exp,extension,semantics) = {
    arith   -> spaces a=atom tail^(a);
    arith   -> spaces x=extension tail^(x);
    atom    -> i=int | '(' a=exp ')' { a };
    int     -> n=(['0','9'])+ { semantics.int(n) };
    tail(l) -> spaces o=op r=exp { semantics.binExp(l,o,r) };
    tail(l) -> { l };
    op      -> ('/' | '*')
  };
  semantics = arith.semantics
}
```

A recursive definition is created using `letrec`-expressions. The language module is instantiated by supplying the root-language, some extensions and the semantics. A fixed point is created that allows the language `lang` to be supplied and returned at the same time. The following examples show an empty extension, an independent extension and an extension that recursively refers to the extended language:

```
letrec lang = arithBase.syntax(lang,{},arithBase.semantics)
in intern lang { 10 * 20 } ⟶ 200

letrec lang = arithBase.syntax(lang,extension,arithBase.semantics)
       extension = {start -> 'two' -> { [| 2 |] }}
in intern lang { 10 * two } ⟶ 20

letrec lang = arithBase.syntax(lang,extension,arithBase.semantics)
       extension = {start -> 'double(' e=lang ')' -> { [| ${e} * 2 |] }}
in intern lang { 10 * double(2 * 2) } ⟶ 80
```

A typical use of extensible language modules occurs when new language constructs are added that need to reference existing language constructs. For example, suppose we want to extend the arithmetic language with lambda-functions and let-binding. Both of these constructs are independently useful so we want language modules for them and we want to mix them into `arithBase`. We can go further than this, because extensible language modules provide control over the way that languages are composed so we will show two variations where let-binding can be interleaved with lambda-functions and secondly where it is limited to the top-level of an expression.

Firstly, we define a language module for lambda-functions. We will assume that there are languages defined elsewhere for `var` and `name` that construct variable expressions and legal program names respectively:

```
type LamSemantics(x:Type) = {
  lambda : (str,x) -> x;
  app : (x,x) -> x
}
lambda : {
  syntax : (L(t),L(t),LamSemantics([| t |])) -> L(t);
  semantics : LamSemantics([| t |])
}
lambda = {
  syntax(operator,exp,semantics) = {
    start -> lam | app | var;
    lam -> spaces 'lam' arg=name dot body=exp {semantics.lambda(arg,body)};
    dot -> spaces '.';
    app -> e=operator '(' a=exp ')' { semantics.app(e,a) }
  };
  semantics = {
    lambda(arg,exp) = Lambda([arg],exp);
    app(o,a) = Apply(o,[a])
  }
}
```

There are now two language modules: `arithBase` which provides an extensible basis for arithmetic expressions, and `lambda` that can be added to other base language modules. To extend `arithBase` we need to *tie the knot* using a fixed point. The two languages are composed so that each knows about the other:

```
letrec
  extension = lambda.syntax(lang.atom,lang,lambda.semantics);
  lang = arithBase.syntax(lang,extension,arithBase.semantics)
  in let f = intern lang { lam x . x / 2 }
     in f(100) ⟶ 50
```

Secondly, the language module for local-binding. Suppose that **let** does not exist in XPL, it can be implemented in terms of function application as follows:

```
type BindSemantics = Fun(x:Type) { bind : (str,x,x) -> x }
bind : {
  syntax : (L(t),BindSemantics([| t |])) -> L(t);
  semantics : BindSemantics([| t |])
}
bind = {
  syntax(value,body,semantics) = {
    start -> local | var;
    local -> bind n=name to v=value for b=body { semantics.bind(n,v,b) };
    bind -> spaces 'bind';
    to -> spaces 'to';
    for -> spaces 'for'
  };
  semantics = {
    bind(name,value,body) =
      [| (fun(${name}) ${body})(${value}) |]
  }
}
```

The language module `bind` is an extension that takes an argument `exp` that is the grammar for the value and body of a local binding expression, for example:

```
letrec
  extension = bind.syntax(lang,lang,bind.semantics);
  lang = arithBase.syntax(lang,extension,arithBase.semantics)
in intern lang { bind x to 30 for x * 10 } ⟶ 300
```

Now we can mix the modules together to create a language that allows lambda-functions and local-binding to be interleaved. Notice that, since we have left open an extension point for the operator in an application expression, we can specify the range of variability for operators:

```
letrec
  letLang = bind.syntax(lang,lang,bind.semantics)
  op = { start -> var | '(' o=lang ')' { o } }
  funLang = lambda.syntax(op,lang,lambda.semantics)
  lang = letLang + funLang
  k = 100
in intern lang { bind f to bind g to lam x.x for g for f(k) }
⟶ 100
```

The modules can be mixed in a different way to produce a language where let-binding is limited to the top-level of an expression. This is achieved by supplying a different definition for the `op` language and a different extension point for `letLang`:

```
letrec
  letLang = bind.syntax(funLang,lang,bind.semantics)
  op = { start -> var | '(' o=funLang ')' { o } }
  funLang=lambda.syntax(op,funLang,lambda.semantics)
  lang = letLang + funLang
in intern lang { bind f to bind g to lam x.x for g for f(k) } ⟶ ERROR
```

Finally, we can mix lambda-functions, local-binding and arithmetic expressions. There are a number of possible value domains for arithmetic expressions so this is done as a language factory where the arithmetic semantics is supplied:

```
complete(semantics) =
  letrec
    letLang = bind.syntax(lang,lang,bind.semantics)
```

```
    op = { start -> var | '(' o=lang ')' { o } }
    funLang = lambda.syntax(op,lang,lambda.semantics)
    lang = arithBase.syntax(lang,letLang + funLang,semantics)
  in intern lang {
       bind double to lam x. x * 2 for bind k to 100 for double(k)
     }

complete(arith.semantics) ⟶ 200
complete(rational.semantics) ⟶ {num=200;den=1}
```

This section has shown a pattern for language modules that can be used to define extension points. An extension point in a language module is a parameter in the syntax definition. When the extension point is instantiated, the parameter is supplied with a grammar. Different instantiations can supply different grammars. Fixed points (created via `letrec`) are used to compose multiple extensible language modules in order to create a single recursive grammar.

Note that this is more expressive than exclusively using set-union to compose grammars. Such a mechanism uses the names of productions *within* grammars as join-points: composing two grammars, both with a rule called `r` will result in a new grammar with a single rule called `r` such that references to `r` within the original grammars now all refer to the composed rule. This is to be compared with composition via parameters and `letrec` where grammars are first-class; as such there is no *intrinsic* relationship between the name of a rule (used as a parameter) and the rule itself. Examples of this are shown above where `op`, `lang`, `letLang`, *etc.*, are passed as arguments without any requirement that those names are used as join-points within the grammars.

Furthermore, composition is defined to be asymmetric in the sense that productions from the left grammar take priority over those from the right. This being the case set-union based composition alone would prevent the construction by composition of a language that relies on a mixture of left-based *and* right-based prioritisation. The use of parameters and fixed-points does not suffer from this restriction because component grammars can be defined in terms of functions over extension points where left-right and right-left prioritised grammars can be supplied as argument values as appropriate.

6. Language Module Morphisms

A language module morphism is a function that maps a language module to a new language module. A morphism can be used to add new functionality, or change existing functionality, in a general way. Section 6.1 shows how to add error handling to arithmetic language modules. To do so, the morphism must know about the semantic interface of the module. Section 6.2 uses monads to generalise this interface, and section 6.3 uses a list monad to add non-determinism to an existing language module.

6.1. An Error Language Morphism

We want to add errors to arithmetic expressions so that execution captures division by 0. If this situation occurs then execution of the complete expression is aborted. This requires a non-local goto since the division by 0 may be buried within a larger expression. A standard way to implement non-local goto is to introduce continuations in the expression so that each expression evaluates and passes its result to a continuation.

If division by 0 is encountered then execution is aborted simply by ignoring the current continuation.

A morphism from arithmetic expressions to continuation based expressions does not need to know anything about the semantics of the base language providing that the language module being transformed provides standard semantic operators `int` and `binExp`. The following language module morphism `addError` performs the necessary transformation for any arithmetic language module:

```
type Error(t:Type) = t + str
type ArithLang(t:Type) = {
  syntax : (ExpSemantics([| t |])) -> L(t);
  semantics : ExpSemantics([| int |])
}
type ErrorLang(t:Type) = {
  syntax : (ExpSemantics([| t |])) -> L(t);
  semantics : ExpSemantics([| ((int)->int) -> Error(int) |])
}
addError : (ArithLang(t)) -> ErrorLang(t)
addError(arithL) = {
  syntax = arithL.syntax;
  semantics = {
    int(x) = [| fun(k) k(${arithL.semantics.int(x)}) |];
    binExp(left,op,right) =
      [| fun(k)
          ${left}(fun(x)
           ${right}(fun(y)
            ${if op = '/'
              then [| if y = 0 then 'division by 0'
                      else k(${arithL.semantics.binExp([|x|],op,[|y|])}) |]
              else k(${arithL.semantics.binExp([|x|],op,[|y|])})})) |]
  }
}
```

The morphism can be applied to the `arith` language module as follows (where the result is the un-evaluated continuation passing expression):

```
let contArith = addError(arith)
in let f = intern contArith.syntax(contArith.semantics) {100/5}
   in f(id) ⟶ 20

let contArith = addError(arith)
in let f = intern contArith.syntax(contArith.semantics) {100/0}
   in f(id) ⟶ 'division by 0'
```

As it stands `addError` cannot be applied to the rational language module because the definition of zero-hood is different. The morphism can be generalized slightly and then used to transform both. The key change is to use a predicate `isZero` to check for `0`. This means that the types must be generalised so that the semantics does not require the use of `int` as the value domain:

```
type ArithLang(t:Type) = {
  syntax : (ExpSemantics([| t |])) -> L(t);
  semantics : ExpSemantics([| t |])
}
type ErrorLang(t:Type) = {
  syntax : (ExpSemantics([| t |])) -> L(t);
  semantics : ExpSemantics([| ((t) -> t) -> Error(t) |])
```

```
}
addError : (ArithLang(t),(t) -> bool) -> ErrorLang(t)
addError(arithL,isZero) = {
 syntax = arithL.syntax;
 semantics = {
  int(x) = [| fun(k) k(${arithL.semantics.int(x)}) |];
  binExp(left,op,right) =
    [| fun(k)
        ${left}(fun(x)
         ${right}(fun(y)
          ${if op = '/'
            then [| if ${isZero}(y) then 'division by 0'
                    else k(${arithL.semantics.binExp([|x|],op,[|y|])}) |]
            else k(${arithL.semantics.binExp([|x|],op,[|y|])})})) |]
  }
}

let contArith = addError(arith,[| fun(n) n = 0 |])
in let f = intern contArith.syntax(contArith.semantics) {100/5}
   in f(id) ⟶ 20

let contRational = addError2(rational,[| fun(r) r.num = 0 |])
in let f = intern contRational.syntax(contRational.semantics) {100/5}
   in f(id) ⟶ {num=100;den=5}
```

This section has shown that it is possible to define functions that map from one language module to another. The ability to define all the constituent elements of a language module as first-class values, as supported by XPL, is important as a basis for general purpose morphisms. Clearly the ability to map a language module will depend on the care with which it has been originally defined. Keeping syntax and semantics separate helps this and a disciplined use of semantic constructor functions in the semantics record makes it easier to define language module morphisms that can be applied to multiple languages.

6.2. Language Modules and Monads

It is difficult to write very general morphisms because the semantics differ from module to module. An aim of Software Language Engineering is to produce reusable language modules where the syntax and the semantics can be reused by combining modules to produce different languages. Unfortunately, in most cases combining different language modules requires the semantics of the individual modules to be modified. Error handling in the previous section is an example where the semantics must be changed to introduce continuations. Other examples, of semantic modification are error values, jumps, backtracking, roll-back and non-determinism.

In many cases changes to the execution of a language can be performed in a general way providing that the language module conforms to a standard structure. Such a general structure is a *monad* [46] [47]. This section extends the structure of an extensible language module with a monad and shows how the arithmetic language modules can be built using monads. It then shows that the execution semantics for the languages can be modified by applying a general purpose monad transformer that has no specific knowledge of arithmetic language modules.

A monad M is a package of two operations `unit` and `bind`. The `unit` operation takes a value v from the language and transforms it into a value *in the monad M(v)*. The bind

operation takes a value in the monad *M(v)* and a function *f*, it supplies *v* to *f* and expects *f* to return another value *M(w)* in the monad. The general structure of an extensible language module including a monad is as follows:

```
lang = {
  syntax(root,extension,semantics) = {
    start -> ...;
    ... grammar rules
  };
  monad = {
    unit(x) = ...
    bind(x,f) = ...
  };
  semantics(monad) = {
    syntaxCnstr(args...) = ...;
    ... syntax constructors defined using monad
  }
}
```

The identity monad makes no changes to the underlying semantic values:

```
idMonad : { unit : (a) -> a; bind : (a,(a)->b) -> b }
idMonad = { unit(x) = x; bind(x,f) = f(x) }
```

The arithmetic language where the underlying value type is integer can be expressed using monads as follows:

```
arithM = {
  syntax =  arithBase.syntax;
  monad = idMonad;
  semantics(monad) = {
    binExp(left,op,right) = [|
      import ${monad} {
        bind(${left},fun(x)
          bind(${right},fun(y)
            unit(${BinExp([| x |],op,[| y |])})))
      }
    |];
    int(cs) = [| import ${monad} { unit(${Int(cs)}) } |]
  }
}
```

Notice in the definition given above that the package of semantics operations is defined in terms of the monad and that the monad is supplied to the package. This means that we can change the language module by replacing the monad and therefore we can get a different semantics with a minimal change. The following shows a use of the `arith` language module:

```
letrec lang = arithM.syntax(lang,{},arithM.semantics([| arithM.monad |]))
  in intern lang {10*20}
⟶ idMonad.bind(idMonad.unit(10),fun(x)
     idMonad.bind(idMonad.unit(20),fun(y)
       idMonad.unit(x * y))
⟶ 200
```

The same upgrade can be applied to the rational language module:

```
rationalM = {
  syntax = arithM.syntax;
```

```
  monad = idMonad;
  semantics(monad) = {
    binExp(left,op,right) =
      case op {
        '*' ->
         [| import ${monad} {
              bind(${left},fun(x)
                bind(${right},fun(y)
                  unit({ num=x.num * y.num; den=x.den * y.den })))
            }  |];
        '/' ->
         [| import ${monad} {
              bind(${left},fun(x)
                bind(${right},fun(y)
                  ${rationalM.semantics(monad).binExp([| unit(x) |],'*',
                     [| unit({ num=y.den;den=y.num }) |])}))
            } |]
       };
    int(cs) = [| import ${monad} { unit({ num=${Int(cs)}; den=1 }) } |]
  }
}

letrec lang = rationalM.syntax(lang,{},rationalM.semantics([| rationalM.
    monad |]))
in intern lang {10/20} ⟶ {num=10;den=20}
```

This approach provides a systematic way of *reifying* control in the semantics component of a language module. The control is supplied by the monad which itself has a standard interface. This allows the monad to be replaced whilst leaving the rest of the language module unmodified. As such this is a mechanism for hijacking the semantics of a language module in a disciplined way.

6.3. The List Monad

The list-monad manages lists of elements. Its `unit` operator creates a singleton list from a value and its `bind` operator maps a function over a list of values. The result of mapping the function produces a list of lists that is then flattened to produce a single-level list.

Lists can be used to represent non-deterministic (ND) computations where the result of an ND computation is any of the elements of the list selected at random. Given any language module, it is possible to transform it into an ND-version by lifting each basic value using `unit` and passing lists of values around. This is done by the following language module morphism:

```
ND(L) = {
  syntax(exp,extension,semantics) =
    L.syntax(
      exp,
      extension,
      semantics);
  monad = {
    bind(x,f) = flatten(map(fun(v) L.monad.bind(v,f),x));
    unit(x) = [L.monad.unit(x)]
  };
  semantics(monad) = L.semantics(monad)
}
```

The morphism can be used on `arithM` as follows:

```
letrec
  syntax = ND(arithM).syntax
  semantics = ND(arithM).semantics
  monad = [| ND(arithM).monad |]
  lang = syntax(lang,{},semantics(monad))
in intern lang {100*2*3}
⟶ [600]
```

As it stands, `arithM` provides no way for the programmer to introduce non-determinism. However, `arith` was defined as an extensible language module and therefore we can introduce a ND language construct as follows:

```
ND(L) = {
  syntax(exp,extension,semantics) =
    L.syntax(
      exp,
      { start ->
          '<' x=exp ',' y=exp '>'
          { semantics.values(x,y) }
      } + extension,
      semantics);
  monad = {
    bind(x,f) = flatten(map(fun(v) L.monad.bind(v,f),x));
    unit(x) = [L.monad.unit(x)]
  };
  semantics(monad) =
    L.semantics(monad) +
    { values(x,y) = [| append(${x},${y}) |] }
}
```

Now it is possible to introduce non-deterministic values:

```
letrec
  syntax = ND(arithM).syntax
  semantics = ND(arithM).semantics
  monad = [| ND(arithM).monad |]
  lang = syntax(lang,{},semantics(monad))
  in intern lang {<100,400>*2*<3,4>}
⟶ [600,800,2400,3200]
```

The monad transformer can be used on a different language module:

```
letrec
  syntax = ND(rationalM).syntax
  semantics = ND(rationalM).semantics
  monad = [| ND(rationalM).monad |]
  lang = syntax(lang,{},semantics(monad))
in intern lang { <100,400> * 2 * <3,4> }
⟶ [{num=600;den=1},{num=800;den=1},{num=2400;den=1},{num=3200;den=1}]
```

To show that `ND` does not rely on the structure of its argument we use it twice:

```
letrec
  syntax = ND(ND(arithM)).syntax
  semantics = ND(ND(arithM)).semantics
  monad = [| ND(ND(arithM)).monad |]
  lang = syntax(lang,{},semantics(monad))
in intern lang { <100,400> * 2 * <3,4> }
⟶ [[600],[800],[2400],[3200]]
```

Errors are still present in even though some of the results are valid. For example, the following fails because of the 0 even though the other results are valid:

```
letrec
  syntax = ND(arithM).syntax
  semantics = ND(arithM).semantics
  monad = [| ND(arithM).monad |]
  lang = syntax(lang,{},semantics(monad))
in intern lang {(<100,400>/<2,4>)/<0,4>}
⟶ ERROR!
```

The semantics can be extended to filter out the illegal values:

```
NDArith(L) = {
  syntax = ND(L).syntax;
  monad = ND(L).monad;
  semantics(monad) =
   let binExp = ND(L).semantics(monad).binExp
   in { binExp(l,op,r) =
          if op='/'
          then binExp(print(l),op,[| remove(0,${r}) |])
          else binExp(l,op,r) } +
       ND(L).semantics(monad)
}
```

Now the transformation preserves only legal values:

```
letrec
  syntax = NDArith(arithM).syntax
  semantics = NDArith(arithM).semantics
  monad = [| NDArith(arithM).monad |]
  lang = syntax(lang,{},semantics(monad))
in intern lang {(<100,400>/<2,4>)/<0,4>}
⟶ [12,6,50,25]
```

The list monad is an example of something can be applied to a language module without knowledge of its implementation. Not all monad transformations are so general, many may require implementation knowledge to be effective. However, these encodings in XPL have shown that given some basic features, it is possible to engineer languages in a way that allows them to be extended and transformed.

7. Implementation

XPL is implemented in Java and consists of a parser and an expression interpreter. The parser is implemented as a machine that manages a stack of choice points in order to perform a parse using a similar mechanism to that of DCG in Prolog (XPL implements a form of cut, !, in grammars although it is not used in this article). The expression interpreter traverses ASTs with respect to a context containing identifier bindings. The parser and interpreter are interleaved to support language embedding. XPL is bootstrapped by creating an XPL grammar as an AST by instantiating the appropriate AST Java classes and then by loading in a file that contains the XPL grammar as a concrete syntax definition. The implementation provides a REPL that reads an XPL command and evaluates it.

Each XPL file is a *module* that contains a collection of definitions. The names defined in a module are private unless they are exported in which case they are available to

anywhere the module is imported. The command `import <path>;` imports a module into the REPL.

The implementation of XPL used to run all the examples in this article is available at: `http://www.eis.mdx.ac.uk/staffpages/tonyclark/Software/xpl.jar`. The file `examples.xpl` contains the 88 examples. The following transcript shows the installation procedure on a Mac together with the instruction for loading the examples file followed by invoking the first two examples. Input to a shell is shown after $ and commands typed to XPL are shown after >:

```
$ mkdir src
$ mv xpl.jar src
$ cd src
$ jar -xvf xpl.jar
 inflated: ...
$ cd ..
$ java -cp src xpl.Interpreter
[src/xpl/xpl.xpl 3231 ms,197]
> import 'src/xpl/examples.xpl';
  [src/xpl/examples.xpl 5506 ms,1569]
  [src/xpl/exp.xpl 23 ms,404]
  [src/xpl/xpl.xpl 1731 ms,170]
  [src/xpl/lists.xpl 106 ms,166]
> example1();
3
> example2();
{x=4;y=6}
>
```

The implementation is organised as a collection of Java packages. The package `Exp` defines the AST classes each of which defines an `eval` method. Also of interest is the file `xpl/xpl.xpl` that defines the concrete syntax of XPL as an XPL grammar.

8. Conclusion and Further Work

There are many tools that support *Software Language Engineering*. Whilst they are all different, the tools have many features in common including the ability to define syntax extensions, synthesize and transform syntax, and embed languages in a host. Most SLE tools focus on defining syntax rather than semantics and few address the issue of modularity.

XPL is a simple functional language extended with features necessary to support SLE. In particular XPL has syntax construction and language embedding. This article has advocated an approach to SLE modularity that treats language definitions as first-class values and has identified the following categories of language module definition and manipulation: independent; substitution; extension; combination; templates; morphisms. The core features of XPL have been validated through a series of examples of increasing abstraction leading to modular homogeneous language embeddings.

Figure 2 shows the key approaches of language definition that are used in this article where the named components are given as XPL definitions in section 3.2 onwards. The figure is divided into five horizontal language categories where the level of reuse increases from top to bottom. Grammars are first-class values in XPL and are an essential part of language modularization. The grammar `arithExternal` is shown with a dashed outline because it is an example of an *external* language that cannot be *embedded*.

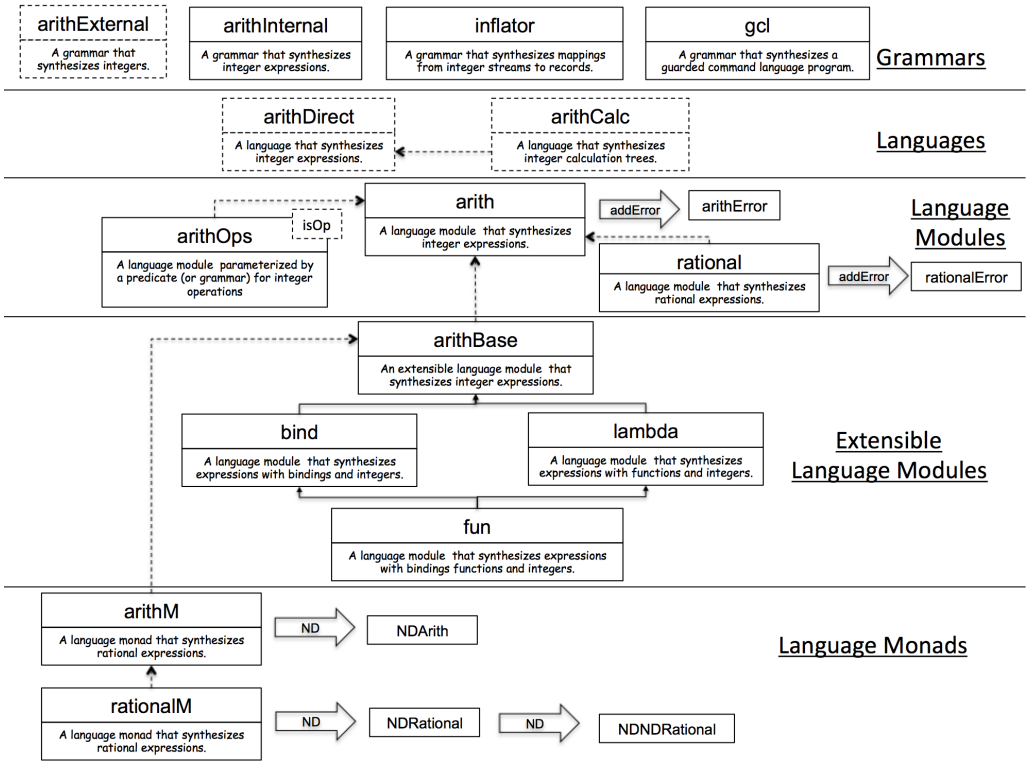

Figure 2: Language Abstraction Levels

XPL defined *languages* are constructed by separating syntax and semantics. A language cannot necessarily be *embedded* and the languages `arithDirect` and `arithCalc` are highlighted as such. A dashed line between languages indicates that some part of the target is used in the definition of the source. This shows that XPL supports language *reuse*.

A *language module* is a language that can be embedded. The language module `arith` is defined as the basis of a number of other languages. An arrow, such as that labelled `addError`, designates a language module morphism, *i.e.*, a mapping from one language module to another. The language modules `arithError` and `rationalError` are both produced by applying the morphism `addError` to different language modules.

A language module may be *parametric* as indicated by a dashed box on the right-hand corner that contains the names of the arguments. The parametric module `arithOps` takes a predicate that determines the syntax of integer binary operators.

Extensible language modules act like super-classes in object-oriented programs. They can be extended with extra language features that add both syntax and semantics to a base language. The language module `arithBase` is extended in two different ways to produce a language of arithmetic expressions with local bindings (`bind`) and a language with arithmetic expressions and higher-order functions (`lambda`). A form of *multiple inheritance* is used to subsequently extend these two languages to produce a language

with arithmetic expressions, local bindings and functions (`fun`).

Language monads are a slightly generalized form of extensible language module whereby the semantics component follows a pattern. Since the semantics follows a standard pattern, it is easier to define language morphisms. The languages `arithM` and `rationalM` are both language monads whose standard structure is exploited by the morphism `ND` that adds non-determinism features to a language. The monadic structure of the languages is exploited to good effect since `ND` is applied to both languages and can even be applied twice as shown in the case of `rationalM`.

There are several directions for this work to explore. Language modules are highly expressive and can be packaged up in various ways. Patterns of usage can be given special syntax that will allow the system to statically check well-formedness. For example, the use of parameters and fixed points in language templates is very similar to the functional encoding of object-oriented systems which leads to the possibility for a definition of language module *inheritance*.

Monads have been used separately to define parsing combinators and to define language execution mechanisms (see for example [48, 44, 49, 50]), however XPL integrates these features in a single language. The use of monads as a basis for modular language definition is a fruitful area for future research.

Grammar modularity and combination is another area whose research challenges are currently open and being actively addressed as described in [51] where the authors extend context-free grammars with modularity features including import and export of non-terminals. The grammars in XPL are equivalent to the simple parser combinators used as stand-alone libraries in functional languages and the combination of XPL grammars is essentially the same as those of MontiCore [38], however the implementation is tightly integrated with the XPL expression evaluator. Furthermore, grammars and grammar productions are first-class values in XPL and can be called as though they are independent functions. This ability makes it possible to link grammars together. XPL provides a cut (`!`) operator whose semantics is the same as that of Prolog in terms of DCGs. This provides a degree of control over the combination of grammars with overlapping rules, although this is an area for further work. XPL does not provide mechanisms for renaming or removing grammar rules and this is an omission that will be addressed in future work in addition to notions of import and export [51].

XPL grammars are closed under combination and, as described above are closely related to recursive descent scannerless parsing. The aim of the XPL implementation has been to experiment with and verify the facilities for language definition and extension, and therefore there has been no investigation of the efficiency of the parsing mechanism. The approach is sufficiently close to Packrat Parsing [52] without being identical for this to be an interesting area for investigation.

XPL is an untyped language. There are significant challenges to integrating static type systems with homogeneous language embedding, not least because of polymorphism and the dependent nature of the resulting types. For example, consider a function like `createAccessor` in this article that maps a string to a definition of a record field accessor. In such a situation, the *type* of the resulting expression depends on the *value* supplied at a different level of processing. Even worse, if the meta-level processing introduces named definitions then there is an implication that types can exist on more than one level. Our motivation in this article is to establish the key properties and patterns of usage of a language for the modular integration of syntax processing and program evaluation. As

such, XPL is unrestricted and it is possible, as in the case of the untyped λ-calculus, to introduce errors that would otherwise be forbidden by a type system. Clearly, this is an important area for further work.

Although XPL has been implemented, it is not tool supported either in terms of itself or in terms of the languages implemented in it. A number of SLE tools, for example XText, provide tool support in the form of syntax highlighting, keyword completion and type checking for the languages that they implement. Whilst this is a secondary concern with respect to XPL, it is important since there may be patterns of language construction that can be exemplified within XPL that have general use across multiple SLE platforms. This is an area for future research.

XPL does not natively support *hygiene* which is the ability for an SLE technology to provide control over where identifiers are bound and thereby to prevent accidental capture of identifier scopes. The foundations of hygiene were developed in the transition from Lisp macros to Scheme syntax processing and could be imported into XPL directly as part of the **intern** construct. However, the aim of XPL is to provide the features of SLE as *first-class* citizens. A possibility is to reify the notion of binding scope as part of XPL and provide programmer control so that standard hygiene is one possible encoding of first-class binding scopes.

References

[1] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[2] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys (CSUR) 37 (4) (2005) 316–344.

[3] M. Eysholdt, H. Behrens, Xtext: Implement your language faster than the quick and dirty way, in: Proceedings of the ACM International Conference Companion on Object-Oriented Programming Systems Languages and Application, ACM, 2010, pp. 307–309.

[4] M. Voelter, Language and ide modularization and composition with mps, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), GTTSE, Vol. 7680 of Lecture Notes in Computer Science, Springer, 2011, pp. 383–430.

[5] J. R. Cordy, TXL - A Language for Programming Language Tools and Applications , Electronic Notes in Theoretical Computer Science 110 (2004) 3–31.

[6] M. Bravenboer, K. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming 72 (1-2) (2008) 52–70.

[7] G. L. Steele, E. E. Allen, D. Chase, C. H. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, Fortress (Sun HPCS Language), in: D. A. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2011, pp. 718–735.

[8] T. Parr, The definitive ANTLR reference: building domain-specific languages, Pragmatic Bookshelf, 2007.

[9] V. Kodaganallur, Incorporating language processing into Java applications: a JavaCC tutorial, Software, IEEE 21 (4) (2004) 70 – 77.

[10] D. A. Ladd, J. C. Ramming, A*: A language for implementing language processors, IEEE Transactions on Software Engineering 21 (11) (1995) 894–901.

[11] E. Moggi, W. Taha, Z.-E.-A. Benaissa, T. Sheard, An idealized metaml: Simpler, and more expressive, in: S. D. Swierstra (Ed.), ESOP, Vol. 1576 of Lecture Notes in Computer Science, Springer, 1999, pp. 193–207.

[12] T. Sheard, S. Peyton-Jones, Template meta-programming for Haskell, in: Proc. Haskell workshop 2002, ACM, 2002.

[13] C. Brabrand, M. I. Schwartzbach, Growing languages with metamorphic syntax macros, in: PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ACM, New York, NY, USA, 2002, pp. 31–40.

[14] R. Lämmel, C. Verhoef, Cracking the 500-language problem, IEEE Software 18 (6) (2001) 78–88.

[15] P. Klint, R. Laemmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering Methodology 14 (3) (2005) 331–380.

[16] M. P. Ward, Language-oriented programming, Software — Concepts and Tools 15 (4) (1994) 147–161.
[17] J. Baker, W. C. Hsieh, Maya: multiple-dispatch syntax extension in Java, in: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 2002, pp. 270–281.
[18] M. Tatsubori, S. Chiba, K. Itano, M.-O. Killijian, OpenJava: A Class-Based Macro System for Java, in: W. Cazzola, R. J. Stroud, F. Tisato (Eds.), Reflection and Software Engineering, Vol. 1826 of Lecture Notes in Computer Science, Springer, 1999, pp. 117–133.
[19] G. L. Steele, Jr., Common LISP: the language (2nd ed.), Digital Press, Newton, MA, USA, 1990.
[20] M. Sperber, R. k. Dybvig, M. Flatt, A. Van straaten, R. Findler, J. Matthews, Revised6 report on the algorithmic language scheme, Journal of Functional Programming 19 (S1) (2009) 1–301.
[21] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (2004) 355–371.
[22] L. Cardelli, F. Ma, T. M. Abadi, Extensible grammars for language specialization, in: In Proceedings of the Fourth International Workshop on Database Programming Languages, Springer-Verlag, 1993, pp. 11–31.
[23] E. V. Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: An extensible attribute grammar system, Science of Computer Programming 75 (1-2) (2010) 39–54.
[24] A. Schwerdfeger, E. V. Wyk, Verifiable composition of deterministic grammars, in: M. Hind, A. Diwan (Eds.), PLDI, ACM, 2009, pp. 199–210.
[25] M. Bravenboer, E. Visser, Parse table composition, in: D. Gasevic, R. Lämmel, E. V. Wyk (Eds.), SLE, Vol. 5452 of Lecture Notes in Computer Science, Springer, 2008, pp. 74–94.
[26] A. Granicz, J. Hickey, Phobos: A front-end approach to extensible compilers, in: HICSS, 2003, p. 324.
[27] L. C. L. Kats, E. Visser, G. Wachsmuth, Pure and declarative syntax definition: Paradise lost and regained, in: Proceedings of Onward! 2010, ACM, 2010.
[28] D. Herman, M. Wand, A theory of hygienic macros, in: S. Drossopoulou (Ed.), ESOP, Vol. 4960 of Lecture Notes in Computer Science, Springer, 2008, pp. 48–62.
[29] T. Clark, L. Tratt, Formalizing homogeneous language embeddings, Electronic Notes in Theoretical Computer Science 253 (7) (2010) 75–88.
[30] M. Berger, L. Tratt, Program logics for homogeneous meta-programming, in: E. M. Clarke, A. Voronkov (Eds.), LPAR (Dakar), Vol. 6355 of Lecture Notes in Computer Science, Springer, 2010, pp. 64–81.
[31] C. Wende, N. Thieme, S. Zschaler, A role-based approach towards modular language engineering, in: M. van den Brand, D. Gasevic, J. Gray (Eds.), SLE, Vol. 5969 of Lecture Notes in Computer Science, Springer, 2009, pp. 254–273.
[32] P. Klint, R. Laemmel, C. Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering and Methodology (TOSEM) 14 (3) (2005) 331–380.
[33] M. Tomita, Efficient parsing for natural language: a fast algorithm for practical systems, Kluwer Academic Pub, 1985.
[34] J. Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (1970) 94–102.
[35] R. Grimm, Better extensibility through modular syntax, in: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2006, pp. 38–51.
[36] D. Spinellis, Notable design patterns for domain-specific languages, Journal of Systems and Software 56 (1) (2001) 91–99.
[37] M. Voelter, K. Solomatov, Language modularization and composition with projectional language workbenches illustrated with mps, Software Language Engineering, SLE (2010) 16.
[38] H. Krahn, B. Rumpe, S. Völkel, Monticore: Modular development of textual domain specific languages, in: R. F. Paige, B. Meyer (Eds.), TOOLS (46), Vol. 11 of Lecture Notes in Business Information Processing, Springer, 2008, pp. 297–315.
[39] C. Wende, N. Thieme, S. Zschaler, A role-based approach towards modular language engineering, in: Software Language Engineering, Springer, 2010, pp. 254–273.
[40] M. Bravenboer, E. Visser, Designing syntax embeddings and assimilations for language libraries, in: H. Giese (Ed.), MoDELS Workshops, Vol. 5002 of Lecture Notes in Computer Science, Springer, 2007, pp. 34–46.
[41] W. Cazzola, I. Speziale, Sectional domain specific languages, in: Proceedings of the 4th workshop on Domain-specific aspect languages, ACM, 2009, pp. 11–14.
[42] T. Clark, L. Tratt, Language Factories, in: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN

Conference Companion on Object-Oriented Programming Systems Languages and Applications, ACM, New York, NY, USA, 2009, pp. 949–955.
[43] G. Hutton, E. Meijer, Monadic parsing in haskell, Journal of functional programming 8 (04) (1998) 437–444.
[44] D. Devriese, F. Piessens, Explicitly recursive grammar combinators - a better model for shallow parser dsls, in: R. Rocha, J. Launchbury (Eds.), PADL, Vol. 6539 of Lecture Notes in Computer Science, Springer, 2011, pp. 84–98.
[45] P. J. Landin, Calculations, Higher-Order and Symbolic Computation 22 (4) (2009) 333–359.
[46] S. Liang, P. Hudak, Modular denotational semantics for compiler construction, in: H. R. Nielson (Ed.), ESOP, Vol. 1058 of Lecture Notes in Computer Science, Springer, 1996, pp. 219–234.
[47] J. Labra Gayo, M. Luengo Díez, J. Cueva Lovelle, A. Cernuda del Río, Lps:: A language prototyping system using modular monadic semantics, Electronic Notes in Theoretical Computer Science 44 (2) (2001) 110–131.
[48] G. Hutton, E. Meijer, Monadic parser combinators, Journal of Functional Programming 8 (4) (1996) 437–444.
[49] J. L. Gayo, M. JMCL, A. del Rio, Reusable monadic semantics of object oriented programming languages, in: Proceeding of 6th Brazilian Symposium on Programming Languages, SBLP02, 2002.
[50] J. Labra Gayo, J. Cueva Lovelle, M. Luengo Díez, A. Cernuda del Río, Specification of logic programming languages from reusable semantic building blocks, Electronic Notes in Theoretical Computer Science 64 (2002) 220–233.
[51] A. Johnstone, E. Scott, M. van den Brand, LDT: a language definition technique, in: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, ACM, 2011, p. 9.
[52] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: ACM SIGPLAN Notices, Vol. 37, ACM, 2002, pp. 36–47.